Systèmes de fichiers distribués sécurisés

Pascal Véron
Groupe de Recherche en Informatique et Mathématiques, Université de Toulon-Var
veron@univ-tln.fr

Résumé

Malgré les nombreux protocoles permettant de partager des fichiers via le réseau, NFS *reste de nos jours le plus largement déployé de par sa simplicité, sa disponibilité sur différentes plates-formes et sa transparence d'utilisation. En contrepartie, les mécanismes de sécurité proposés par* NFS *sont pratiquement inexistants. L'objet de ce document n'est pas de discuter de protocoles alternatifs mais de présenter différentes méthodes permettant d'utiliser un serveur* NFS *opérationnel de façon sécurisée. Dans un premier temps, nous montrerons comment renforcer la sécurité de* NFS *en utilisant le tunneling* SSH*. Nous présenterons ensuite quelques systèmes qui se placent en amont de* NFS *et utilisent des algorithmes cryptographiques robustes pour sécuriser les échanges et authentifier les utilisateurs. Pour chaque cas étudié, nous montrerons comment mettre en place un accès sécurisé entre un poste client se trouvant sur un réseau public et un serveur* NFS *situé dans un réseau privé via une passerelle* SSH*.*

Mots clefs

CryptoFS, Nfs, Rpc, Sec Rpc, Sfs, Shfs, Ssh, Tcfs, Partage sécurisé, Tunneling.

1 Introduction

Développé dans les années 80 par la société SUN MICROSYSTEMS, le protocole NFS en est actuellement à sa version 3 [1] (la version 4 [2] devrait apparaître très prochainement et apportera de nombreuses modifications qui devraient rendre le reste de ce document plus ou moins obsolète si toutes les spécifications requises sont effectivement opérationnelles et si l'ensemble des serveurs et clients de la planète migrent vers cette dernière version). On peut considérer que, dans le milieu UNIX, NFS est actuellement le protocole de partage de fichiers le plus utilisé. Il possède cependant de nombreuses lacunes en ce qui concerne la sécurité :

- les données transitent en clair sur le réseau entre le client et le serveur,
- aucun contrôle sur l'intégrité des données échangées n'est effectué (les données peuvent donc être interceptées et modifiées lors de leurs parcours),
- hormis le contrôle de l'adresse IP, le poste client n'a aucun moyen d'authentifier l'identité du serveur NFS (et vice-versa),
- le contrôle de l'accès aux données se fait uniquement par vérification du couple *(uid,gid)* associé à la requête. Le serveur ne dispose d'aucun moyen pour vérifier que ce couple correspond bien à l'émetteur de la demande. Un administrateur système peu scrupuleux peut créer sur son poste de travail des utilisateurs fictifs ayant pour *uid* et *gid* des valeurs lui permettant d'accéder aux données situées sur un serveur NFS, il ne lui reste plus qu'à usurper l'adresse IP d'un client autorisé pour pouvoir effectivement accéder à ces dernières.
- les utilisateurs doivent avoir toute confiance en l'administrateur système du serveur NFS.

Les différentes méthodes présentées dans ce document (tunneling SSH, SECRPC, SHFS et SFS) apportent toutes une solution aux deux premiers points et assurent l'authentification du serveur par le poste client. Seul SFS utilise le principe de la signature numérique pour certifier chaque requête. Le chiffrement des données sur le serveur peut apporter une solution au dernier point, nous y reviendrons lorsque nous évoquerons les systèmes CRYPTOFS [3] et TCFS [4] . A l'exception de TCFS, toutes nos expérimentations ont été effectuées sur des machines LINUX pourvues d'un noyau de la série 2.4.x (redhat 7.3 ou 8.0).

Cependant les solutions proposées doivent pouvoir s'adapter sans problème sous d'autres systèmes Unix. Nous avons orienté notre recherche vers des solutions rapides à mettre en œuvre, nécessitant une intervention minimum de la part de l'administrateur système et pour lesquelles -si possible- il n'est pas utile de recompiler le noyau. Chaque solution est étudiée dans deux contextes particuliers (cf. fig. 1) :

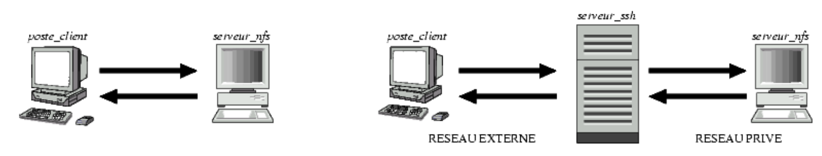


Figure 1 – *Contexte de travail*

- *Accès direct* : le poste client (nommé *poste_client*) et le serveur NFS (nommé *serveur_nfs*) sont reliés via le réseau,
- *Accès indirect* : le serveur NFS se trouve sur un réseau privé et le poste client sur un réseau externe. Le réseau privé dispose d'un serveur SSH (nommé *serveur_ssh*) relié au réseau externe. Seul le flux entrant à destination du port 22 et le flux sortant issu du port 22 sont autorisés sur l'interface externe de cette machine.

Afin de mieux aborder les solutions présentées, nous commencerons dans une première partie par rappeler quelques notions indispensables concernant le protocole NFS et les RPC. Nous décrirons ensuite le mécanisme du tunneling SSH. Nous détaillerons alors les différentes solutions retenues et nous terminerons par une analyse de l'impact de l'ajout de mécanismes de sécurité sur les performances générales.

2 NFS et RPC

NFS s'appuie sur le protocole RPC (Remote Procedure Call, développé par Sun [5]) qui de façon non formelle permet à un programme (le client RPC) d'exécuter une fonction d'un programme (le serveur RPC) en exécution sur une autre machine. Les messages RPC (appel à une fonction) sont envoyés en mode TCP ou UDP. NFS utilise UDP par défaut. Quand un serveur RPC s'exécute (par exemple le démon `nfsd` ou `mountd`) , il s'enregistre sur la machine auprès d'un programme spécial -le portmapper- qui recense tous les serveurs RPC disponibles et assigne à chacun d'entre eux un numéro et un port d'écoute. Ces informations sont consultables via la commande `rpcinfo -p`. A titre d'exemple voici un extrait du résultat obtenu sur une machine linux hébergeant un serveur NFS :

```
100005    3   udp   1025  mountd
100005    3   tcp   1025  mountd
100003    3   udp   2049  nfs
```

La version 3 de `mountd` est en écoute en mode TCP et UDP sur le port 1025 et correspond au numéro 100005. Le démon `nfsd` est en écoute en mode UDP sur le port `2049`.

Lorsqu'un client RPC veut envoyer une requête vers un serveur RPC, il interroge le portmapper du serveur distant afin d'obtenir le port d'écoute du serveur concerné. Parler du protocole NFS est un abus de langage. NFS se compose en fait de deux protocoles distincts : MOUNT et NFS. MOUNT intervient lors de la négociation initiale entre le serveur et le client. Il permet de déterminer quelles partitions sont exportées et fournit au client un descripteur de fichiers utilisé par la suite pour accéder à la racine du système de fichiers exporté. NFS prend alors le relais et gère toutes les demandes d'accès (écriture, lecture). La syntaxe générale de la commande `mount` est la suivante :

`mount [-omountport=....,port=....]` *hote*:*partition point_de_montage*

La partie `hôte` indique à la commande `mount` où se trouve le portmapper à interroger. Ce dernier renvoie une liste des serveurs RPC enregistrés sur *hôte*. De cette liste, `mount` extrait les ports p_1 et p_2 associés aux programmes 100005 et 100003 qui correspondent à `mountd` et `nfsd`. Une requête de demande de montage de la partition distante est alors envoyée en UDP sur le port p_1 de *hôte*. Les accès à la partition sont effectués en envoyant des requêtes sur le port p_2 de *hôte*. Dans le cas où les options `mountport` et `port` sont spécifiées ce seront ces valeurs qui seront utilisées pour les requêtes RPC (l'appel initial au portmapper distant est dans tous les cas effectué).

Sans aucune authentification, n'importe quel client pourrait envoyer des requêtes RPC à un serveur et obtenir sa réponse. Le serveur RPC doit pouvoir disposer d'un mécanisme pour savoir s'il peut délivrer ou non des informations au client l'interrogeant. Il existe 4 systèmes d'authentification de base développés par Sun :

- AUTH_NONE : l'accès est anonyme, c'est au serveur de déterminer quelles informations sont accessibles pour ce type d'accès,
- AUTH_UNIX ou AUTH_SYS : l'uid et le gid du demandeur sont envoyés au serveur qui vérifie en fonction de ces valeurs si la requête est autorisée,
- AUTH_DES (non disponible sous linux) : utilise un mécanisme classique combinant la cryptographie à clé publique et à clé privée permettant à l'émetteur de la requête et au serveur de partager une clé de chiffrement commune. Nous ne détaillerons pas le mécanisme d'authentification, mais nous insisterons sur trois problèmes essentiels :
  - la clé commune est mise en place avec l'algorithme de Diffie-Helmann [6]. La sécurité de ce dernier repose sur un problème arithmétique difficile : le logarithme discret [7]. Étant donné les progrès actuels concernant la recherche de solutions à ce problème, les entiers manipulés pour assurer la sécurité du protocole doivent se composer d'au moins 1024 bits. Or l'implémentation de Sun utilise des entiers de 192 bits . . .
  - l'algorithme utilisé pour authentifier les requêtes est le DES [8] avec une clé de 56 bits. Depuis la cryptanalyse différentielle de Biham et Shamir [9], la cryptanalyse linéaire de Matsui [10], et le développement de circuits spécialisés pour attaquer le DES [11], ce protocole est à présent considéré comme obsolète par la communauté scientifique.
  - le mécanisme AUTH_DES ne sert qu'à authentifier les requêtes RPC, les données transmises via ces requêtes ne sont pas chiffrées . . .
- AUTH_KERB : nécessite la mise en place d'un serveur Kerberos, nous n'aborderons pas ce sujet.

3 Redirection de ports avec SSH

Le protocole SSH a été développé en 1995 par T. Ylönen [12]. Actuellement le protocole en est à sa version 2. Il se compose de 3 couches :

- la couche connexion qui se charge de l'authentification du serveur par l'utilisateur, de l'élaboration d'une clé de chiffrement commune pour assurer la confidentialité des données et du contrôle de l'intégrité de ces dernières,
- la couche authentification qui permet au serveur d'authentifier l'utilisateur (et non pas le poste client),
- la couche connexion qui permet en particulier de mettre en place des tunnels sécurisés.

Les phases d'authentification sont assurés par un algorithme à clé publique, généralement RSA [13] ou DSA [14]. Par défaut dans la distribution OPENSSH [15] la clé publique du serveur est de taille 768 bits (comme pour la carte bancaire), mais nous conseillons d'en générer une nouvelle de taille 1024. La mise au point de la clé commune est assurée par l'algorithme de Diffie-Helmann en utilisant des entiers de taille 1024 bits. On pourra consulter la table de Lenstra-Verheul [16] qui indique -en fonction des attaques connues à ce jour- quelle taille de clés doit être utilisée en cryptographie à clé publique pour obtenir une sécurité équivalente au DES en 1982. Le chiffrement des données est effectué par des algorithmes à clé secrète dont la sécurité a été longuement analysé : 3DES [17], AES [18], BLOWFISH [19], ARCFOUR [20], ... Par défaut, c'est l'AES (successeur du DES depuis octobre 2000) avec une clé de 128 bits qui est utilisé. L'utilisation de fonctions de hachage comme SHA-1 [21] ou MD5 [22] permet d'assurer l'intégrité des données.

BLOWFISH et ARCFOUR sont moins "populaires" que 3DES et l'AES car ils sont difficiles à implémenter au niveau matériel. Ils possèdent tous deux un débit de chiffrement élevé. L'algorithme ARCFOUR est un générateur de flux pseudo-aléatoire dont les bits de sortie sont combinés par un *ou-exclusif* avec les bits du message à chiffrer. Cet algorithme est très peu sûr au sens académique du terme, cependant il n'existe aucune attaque réellement pratique quand il est employé correctement au sein d'un protocole bien pensé (comme SSL par exemple). BLOWFISH est un algorithme de chiffrement par blocs comme 3DES et l'AES. Il fait partie de SSH car les concepteurs de ce protocole cherchaient un algorithme libre de droit qui ne soit pas un standard recommandé par le gouvernement américain. Le 3DES est très robuste mais beaucoup plus lent que les autres. Aucune cryptanalyse linéaire ou différentielle applicable à ces 3 algorithmes n'a été trouvée à ce jour.

La redirection de ports consiste à mettre en place un tunnel "chiffré" entre deux applications non sécurisées afin que celles-ci puissent communiquer (en mode TCP) de façon confidentielle. Le manuel de SSH n'étant pas très explicite sur cette fonctionnalité (notamment en ce qui concerne le trajet sur lequel les données seront réellement chiffrées), nous commencerons par brièvement rappeler le déroulement d'une communication sécurisée via un tunnel SSH. La commande permettant la mise en place d'un tunnel SSH est la suivante :

`client% ssh -f -N -L` *port_local*`:`*poste_distant*`:`*port_distant* `[`*login*`@]`*serveur_ssh*

Trois machines sont impliquées lors de la création du tunnel : le poste client, un serveur SSH et le poste distant avec lequel le client va communiquer. Une fois cette commande exécutée sur le poste client :

- le serveur SSH s'assure de l'identité de l'utilisateur à l'origine de cette requête,
- sur le poste client, un client SSH se met en écoute sur *port_local* via l'interface réseau locale (loopback),
- toute requête vers `localhost:`*port_local* est automatiquement transmise de façon confidentielle -via l'interface réseau externe- par le client SSH (génération d'un nouveau numéro de port source) vers le port 22 de *serveur_ssh*,
- le serveur SSH émet alors la requête comme un client normal vers *poste_distant*`:`*port_distant* (*port_distant* reçoit un paquet avec pour port source un port aléatoire engendré par *serveur_ssh* et pour adresse IP source l'adresse IP de *serveur_ssh*),
- *poste_distant* renvoie sa réponse à *serveur_ssh* qui à son tour la retransmet via le tunnel vers le client SSH de *poste_client*,
- le client SSH transmet la réponse reçue vers `localhost:`*port_local*.

La figure 2 illustre ce mécanisme dans le cas où le poste client désire contacter le serveur TELNET du poste distant. Pour chaque requête, sont précisés le port source, la machine émettrice et le port destination.

`client% ssh -f -N -L 2323:`*poste_distant*`:23` *serveur_ssh*

`client% telnet localhost 2323`

Remarques importantes :

- L'option `-N` signifie que l'on ne désire pas exécuter une commande distante sur le serveur SSH, l'option `-f` permet de lancer la commande en tache de fond une fois le mot de passe (ou la phrase clé) saisie pour l'identification.
- Seul l'administrateur système peut mentionner en tant que port local un port privilégié (< 1024).

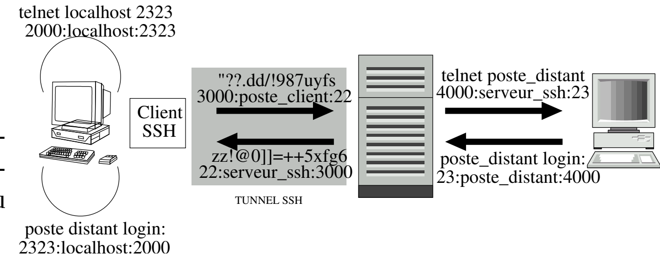


Figure 2 – *Mécanisme de redirection de ports*

- Le tunnel n'est accessible que via l'interface loopback du poste client.
- Pour le poste distant la requête est émise depuis le serveur SSH, il n'a pas connaissance du poste client.

. Le poste distant n'étant contacté que par le serveur SSH, il n'est pas nécessaire de spécifier sur la ligne de commande pour *poste_distant* une adresse IP -ou un nom de machine- directement joignable par le poste client (ceci permet par exemple d'atteindre des machines d'un réseau privé disposant d'une machine bastion hébergeant un serveur SSH).
. Seules les communications entre le poste client et le serveur SSH sont chiffrées. Un intrus peut observer la communication en clair entre le serveur SSH et *poste_distant*.
. Dans le cas où *poste_distant* et *serveur_ssh* sont une seule et même machine, on obtient une communication entièrement sécurisée (ainsi dans le cas où *serveur_ ssh* héberge aussi un serveur *Pop*, ceci permet par exemple de relever son courrier sans que les messages transitent en clair).
. La communication n'est pas chiffrée au niveau de l'interface loopback.
. Le tunnel reste actif tant que le client SSH qui tourne en tache de fond sur *poste_client* n'est pas arrêté par l'utilisateur. Pour que le tunnel disparaisse automatiquement lorsque l'application qui l'utilise se termine, il suffit de modifier la commande initiale comme suit :
`client% ssh -f -L` *port_local*`:`*poste_distant*`:`*port_distant* `[`*login*`@]`*serveur_ssh* `sleep 30`
Le tunnel est créé et la commande `sleep 30` est lancée sur le serveur SSH. Au bout de 30 secondes, le client SSH disparaît ainsi que le tunnel. Cependant si dans ce laps de temps, une connexion TCP est initiée sur le port *port_local* de *poste_client*, le tunnel reste actif jusqu'à la terminaison de la communication. La clôture de la session TCP provoquera automatiquement la destruction du tunnel.
. une fois le tunnel créé, n'importe qui peut l'utiliser.

SSH n'est capable de rediriger que le flux TCP, or le protocole NFS utilise UDP par défaut dans le noyau 2.4.x de Linux. Pour obtenir une version en mode TCP, il est nécessaire de recompiler le noyau. Bien que considéré comme expérimental le serveur en mode TCP fonctionne parfaitement. Si on ne souhaite pas recompiler le noyau, le package `nfs-user-server` pour la distribution Debian propose un serveur NFS en mode TCP qui s'exécute dans l'espace utilisateur. Nous distinguerons donc pour la suite les deux modes de connexion possibles : UDP et TCP. Nous supposerons de plus pour les exemples que le serveur NFS exporte la partition `/home`.
Remarque : FreeBSD et Solaris supportent nativement une version TCP de NFS.

4 Sécuriser un serveur NFS-TCP avec SSH

4.1 Accès direct

Le serveur NFS doit aussi dans ce cas héberger un serveur SSH.
Configuration poste client A l'aide de la commande `rpcinfo` on détermine sur quels ports les démons correspondants aux services MOUNT et NFS sont en écoute sur le serveur :
`client% rpcinfo -p` serveur_nfs
Généralement le démon `mountd` est en écoute sur un port aléatoire déterminé par le portmapper au moment du lancement du serveur, cependant celui-ci peut être fixé une fois pour toutes en utilisant l'option `-p` de `rpc.mountd`. Le démon `nfsd` est par défaut en écoute sur le port 2049. Une fois ces informations obtenues, on peut lancer la création du tunnel pour accéder en mode sécurisé à ces ports :
`client% ssh -f -N -L 25000:`*serveur_nfs*`:1025 -L 25001:`*serveur_nfs*`:2049 [`*login*`@]`*serveur_nfs*
Remarques :
. il n'est pas indispensable que le tunnel SSH soit créé par l'administrateur du système, cependant ce dernier est normalement le seul à pouvoir utiliser la commande `mount`,
. les ports 25000 et 25001 ne sont donnés qu'à titre d'exemple.

Une fois la phase d'identification terminée, pour monter le système de fichiers il suffit d'exécuter (en tant que `root`) :
```
client% mount -t nfs -otcp,mountport=25000,port=25001 localhost:/home /mnt/home
```

Quelques commentaires sur la commande :
. Une requête de montage pour la partition `/home` est envoyée sur le port local 25000 (puisque la commande `mount` s'adresse à `localhost`).
. La requête transite via le tunnel sécurisé jusqu'au serveur SSH.
. Le serveur SSH engendre alors vers le port 1025 du serveur NFS (nous rappelons qu'il s'agit de la même machine) une requête de montage pour la partition `/home`.
. Si la requête est acceptée (cf. configuration du serveur NFS), l'autorisation est renvoyée au serveur SSH qui à son tour la transmet au poste client.
. Pour la suite toutes les requêtes de type NFS sont dirigées vers `localhost:25001` et sont donc transmises vers le port `2049` du serveur NFS.

Pour que l'ensemble fonctionne correctement, il est indispensable que le serveur NFS accepte les requêtes envoyées par le serveur SSH.

Configuration serveur NFS Comme précisé dans le paragraphe 3, les requêtes (de type `MOUNT` ou `NFS`) reçues par le serveur NFS ne proviennent pas du poste client mais du serveur SSH. Outre la vérification de l'uid et du gid du demandeur, le serveur NFS vérifie l'adresse IP source de la requête avant de l'accepter. Il faut donc indiquer dans le fichier `/etc/exports` que la partition `/home` est accessible pour le serveur SSH. Ce qui revient à exporter la partition `/home` du serveur NFS vers lui même puisque les deux serveurs tournent sur la même machine :

```
/home serveur_nfs(rw,root_squash,insecure)
```

Remarques :

. Il est conseillé d'exporter la partition en précisant soit l'adresse IP du serveur soit son nom. Il existe en effet des attaques pour les serveurs exportant des partitions vers `localhost` via l'interface locale.

. Le démon de montage n'accepte que des requêtes issues d'un port réservé, sauf si la directive `insecure` est utilisée. Cette dernière est indispensable car les requêtes émises par le serveur SSH proviennent d'un port aléatoire (cf. §3).

Automatisation de la procédure En rajoutant dans le fichier `/etc/fstab` de poste_client la ligne suivante :

```
localhost:/home /mnt/home nfs tcp,intr,bg,mountport=25000,port=250001
```

la partition sera automatiquement montée au démarrage du système (si le tunnel est déjà en place). Ceci nécessite de la part de l'administrateur système du poste client de s'identifier à chaque redémarrage. Pour éviter cela, l'administrateur peut générer un jeu de clés non liées à un mot de passe ou à une phrase clé : `ssh-keygen -t rsa`.

Lorsque la commande demande de rentrer une phrase clé, il suffit de valider pour engendrer le couple de clés. On supposera pour la suite que la clé publique générée est dans le fichier `id_rsa_tunnel.pub`. L'administrateur de *poste_client* place dans son compte sur le serveur SSH -dans le fichier `authorized_keys` situé dans le répertoire `.ssh/`- le contenu de `id_rsa_tunnel.pub` qui est de la forme `ssh-rsa` *clé publique encodée en base64* `root@`*poste_client*, et rajoute au début de cette ligne les directives :

from="*poste_client*",permit-open="*serveur_ssh*:1025",permit-open="*serveur_ssh*:2049",
command="/bin/sleep 300".

La clé privée n'étant pas protégée par une phrase clé, si un intrus arrivait à la subtiliser il pourrait accéder -via SSH - au compte de l'administrateur système de *poste_client* sur *serveur_ssh* sans aucune identification. La ligne ci-dessus limite le champ d'action d'un tel individu, en effet :

. la directive `host` ne permet l'utilisation de la clé `id_rsa_tunnel` qu'à partir de la machine *poste_client* obligeant l'attaquant à pratiquer aussi l'IP spoofing pour parvenir à ses fins ou à mener l'attaque depuis le poste client,

. la directive `command` lance l'exécution de la commande spécifiée lors d'une connexion SSH utilisant la clé `id_rsa_tunnel`, et ignore toute autre commande spécifiée par le client SSH,

. la directive `permitopen` n'autorise la redirection de ports que vers la (ou les) machine(s) et le (ou les) port(s) distant(s) spécifiés.

Il suffit alors de rajouter sur *poste_client* -dans l'un des scripts exécutés au démarrage et avant la consultation de `/etc/fstab`)- la ligne : `ssh -f -T -i /root/.ssh/id_rsa_tunnel -L 25000:`*serveur_nfs*`:1025 -L 25001:`*serveur_nfs*`:2049 [`*login*`@]`*serveur_nfs* `sleep 10`

Remarques :

. l'option `-i` permet de spécifier la clé à utiliser pour le mécanisme d'identification,

. l'option `-T` signifie que l'on ne désire pas allouer de terminal pour l'exécution de la commande,

. la commande est lancée en tache de fond, cependant l'option `-f` ne peut être utilisée que si l'on spécifie une commande distante à exécuter. C'est pourquoi nous avons rajouté `sleep 10` (qui ne sera jamais exécutée étant donnée que la configuration du fichier `authorized_keys` sur *serveur_ssh* lance l'exécution de `sleep 300` lors de l'utilisation de la clé `id_rsa_tunnel`),

. le tunnel SSH est actif pour 300 secondes, ce qui laisse le temps au client de monter la partition `/home` et de maintenir l'existence du tunnel,

. lors du démontage de la partition, le tunnel sera automatiquement détruit.

4.2 Accès indirect

Configuration du serveur NFS Il faut modifier le fichier `/etc/exports` afin d'exporter la partition `/home` vers le serveur SSH puisque c'est lui qui va retransmettre les requêtes NFS :

```
/home serveur_ssh(rw,root_squash,insecure)
```

Configuration du poste client Il suffit d'exécuter : `ssh -f -N -L 25000:`*serveur_nfs*`:1025 -L 25001:`*serveur_nfs*`:2049 [`*login*`@]`*serveur_ssh*

Selon le contexte de travail, une telle configuration soulève le problème suivant : la communication entre le serveur SSH et le serveur NFS n'est pas sécurisée. Pour remédier à cet inconvénient on peut combiner 2 tunnels pour obtenir une communication sécurisée sur l'ensemble du trajet (si le serveur NFS héberge aussi un serveur SSH et que l'administrateur de *poste_client* possède un compte utilisateur sur le serveur NFS). Nous commencerons par donner une explication générale puis nous détaillerons comment automatiser le procédé en utilisant les fonctionnalités de `ssh-agent`.
L'idée est donc de créer un tunnel de *poste_client* vers *serveur_ssh* et un tunnel de *serveur_ssh* vers *serveur_nfs*. Sur le poste client on exécute : `ssh -f -L 25000:localhost:25001 -L 25002:localhost:25003 [login@]serveur_ssh sleep 30`
et sur le serveur SSH : `ssh -f -L 25001:serveur_nfs:1025 -L 25003:serveur_nfs:2049 [login@]serveur_nfs sleep 30`
On peut alors exécuter sur le client : `mount -otcp,mountport=25000,port=25002 localhost:/home /mnt/home`
. *Remarque :* Sur le client, le fait de spécifier dans la commande `localhost` et non pas *serveur_ssh* est indispensable. En effet, le deuxième tunnel -sur *serveur_ssh* - n'est accessible que via l'interface de loopback sur le port 25001 (cf. §3). En précisant *serveur_ssh* au lieu de `localhost`, toute requête sur le poste client à destination de `localhost:25000` aurait été transmise au serveur SSH qui aurait essayé de les retransmettre sur le port 25001 via l'interface réseau.

La figure 3 illustre le parcours aller-retour d'une requête de type `mount`. Pour automatiser la création des 2 tunnels :
. l'administrateur de *poste_client* ajoute dans le fichier `authorized_keys` (dans son compte sur serveur SSH) le contenu de `id_rsa_tunnel.pub`, et y rajoute les directives :
`from="poste_client",permitopen="localhost:25001",permitopen="localhost:25003",`
`command="ssh -T -L 25001:serveur_nfs:1025 -L 25003:serveur_nfs:2049" [login@]serveur_nfs`
. il fait de même sur le serveur NFS en spécifiant pour directive :
`from="serveur_ssh",permitopen="serveur_nfs:1025",permitopen="serveur_nfs:2049",command="/bin/sleep 300"`

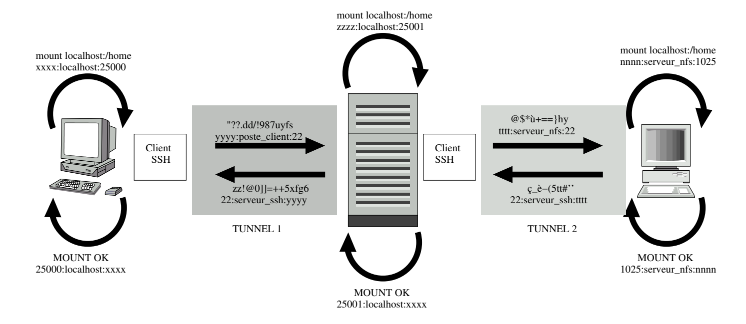


Figure 3 – *Utilisation de 2 tunnels*

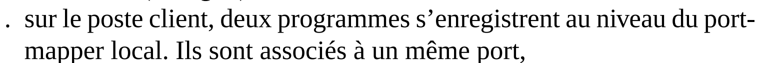
La création du deuxième tunnel nécessite l'authentification d'un utilisateur avec la clé publique `id_rsa_tunnel.pub`. Pour cela, il faut que la clé privée `id_rsa_tunnel` stockée sur *poste_client* soit aussi présente sur *serveur_ssh* dans le répertoire `.ssh/` de l'utilisateur. Cette clé n'étant pas protégée par une phrase clé, il est conseillé d'éviter sa duplication dans des endroits différents. C'est à ce stade qu'intervient `ssh_agent`. Cette commande permet d'enchaîner une succession de connexions SSH d'un serveur sur un autre en utilisant une même clé publique sans qu'il soit nécessaire que la clé privée soit présente sur chacun des comptes traversés. A chaque demande d'identification, la requête est retransmise via les différents clients SSH jusqu'à l'agent présent sur le premier client. Ce dernier ayant accès à la clé privée de l'utilisateur, il prend en charge la requête d'authentification et renvoie la réponse au serveur concerné sans intervention de l'utilisateur. Pour mettre en place ce mécanisme :
. on exécute sur *poste_client* la commande `eval 'ssh-agent'` qui permet de lancer l'agent et de positionner certaines variables d'environnement,
. on fournit à l'agent la clé nécessaire pour l'identification :`ssh-add id_rsa_tunnel`
. on indique lors de la création du premier tunnel que le serveur distant devra utiliser l'agent s'il initie une connexion SSH vers un autre serveur (option `-A` de SSH) :

```
client% ssh -A -f -i id_rsa_tunnel -L 25000:localhost:25001 -L 25002:localhost:25003 \
                                                    [login@]serveur_ssh sleep 30
```

Un intrus qui aurait réussi à subtiliser la clé `id_rsa_tunnel`, peut :
. à partir du serveur SSH, soit lancer l'exécution de la commande `sleep` sur le serveur NFS, soit monter la partition `/home` via un tunnel,
. à partir du poste client, uniquement monter la partition `/home` via un double tunnel,
Pour éviter que la partition `/home` ne puisse être montée, nous conseillons dans ce cas de générer la clé `id_rsa_tunnel` en utilisant une phrase clé. On utilisera l'agent SSH afin de n'avoir à saisir cette dernière qu'une seule fois lors du démarrage du système.
Remarque : Bien que la communication soit à présent chiffrée d'une extrémité à l'autre, il subsiste un dernier petit inconvénient. L'administrateur système du serveur SSH peut se servir de l'existence du deuxième tunnel pour accéder à la partition `/home` du serveur NFS.

5 Sécuriser un serveur NFS-UDP avec SSH

Le logiciel SEC RPC développé par J. Bowmman [23] permet d'utiliser SSH dans le cas d'un serveur NFS en mode UDP. La dernière version (1.52-1) date de Juillet 2003 et se compose de trois packages RPM : `sec_rpc-1.52-1`, `sec_rpc-client-1.52-1` et `sec_rpc-server-1.52-1`. Le premier est à installer à la fois sur le poste client et le serveur NFS. Le fonctionnement de SEC RPC est le suivant (cf. fig. 4) :



- sur le poste client, deux programmes s'enregistrent au niveau du portmapper local. Ils sont associés à un même port,

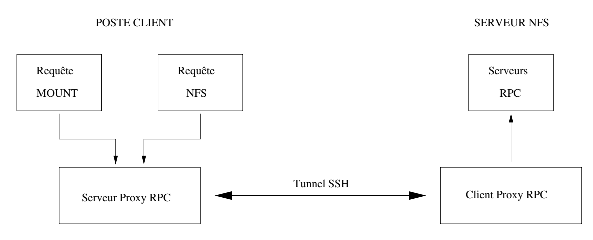


Figure 4 – *Fonctionnement de Sec Rpc*

- en utilisant les directives `mountprog` et `nfsprog` de la commande `mount` on redirige les requêtes de type `MOUNT` et `NFS` vers ces programmes,
- un démon (serveur proxy RPC) en écoute sur le port associé à ces derniers récupère les requêtes UDP, les encapsule dans une trame TCP et les transmet via SSH à un programme en écoute sur le serveur (client proxy RPC),
- celui-ci décode les requêtes et les retransmet comme un client RPC normal au portmapper local (ce qui signifie que pour le serveur toutes les requêtes NFS reçues proviendront en fait de lui-même).

5.1 Accès direct

Configuration du serveur NFS Après installation des packages `sec_rpc-1.52-1` et `sec_rpc-server-1.52-1`, un utilisateur `snfs` est créé et des couples de clés privées/publiques (non protégées par une phrase clé) sont générées dans `/etc/snfs/.ssh`. Le client proxy RPC se nomme `rpc_pcl`. Les requêtes NFS étant transmises du serveur vers lui-même, le fichier `/etc/exports` doit être configuré ainsi :

`serveur_nfs /home(rw)`

Configuration du client Après installation des packages `sec_rpc-1.52-1` et `sec_rpc-client-1.52-1`, un utilisateur `snfs` a été créé. C'est le programme `rpc_psrv` qui se charge de l'enregistrement de nouveaux services dans le portmapper, du lancement du serveur proxy RPC et de l'exécution du client proxy RPC sur le serveur NFS via l'utilisateur `snfs`. Pour cela, il utilise un fichier de configuration situé dans `/etc/snfs`. La création de ce dernier s'effectue en exécutant :

`client% snfshost` *serveur_nfs :MOUNTPROG*.

MOUNTPROG est un entier à choisir entre 200000 et 249999 de façon à ce que *MOUNTPROG* et *NFSPROG*=*MOUNTPROG*+50000 soient deux programmes n'étant pas déjà enregistrés au niveau du portmapper. On supposera pour la suite que *MOUNTPROG*=2000000. Le fichier ainsi créé a pour nom *serveur_nfs*. Il contient les directives nécessaires pour rediriger les appels RPC à destination des programmes associés à *MOUNTPROG* et *NFSPROG* vers le serveur NFS via SSH. On y trouve aussi la ligne de commande permettant de lancer sur le serveur l'exécution du client proxy RPC (`rpc_pcl`). On rajoute dans le fichier `/etc/fstab` la ligne :

poste_client`:/home /mnt/home nfs mountprog=200000,nfsprog=250000,hard,intr,noauto`

Finalement afin que la communication SSH puisse se faire automatiquement entre le client et le serveur, il faut déplacer la clé `id_rsa` -située sur le serveur- de l'utilisateur `snfs` dans son compte sur le poste client. On pourra effacer les clés DSA et renommer le fichier `authorized_keys2` en `authorized_keys`, l'utilisation de SSH V.1. étant déconseillée. Assurez-vous que le répertoire `.ssh/` appartienne à l'utilisateur `snfs`. Vérifiez que l'utilisateur `snfs` peut bien se connecter sans mot de passe sur le serveur. Pour monter la partition, il suffit alors d'exécuter : `/etc/rc.d/init.d/snfs start`

Remarque : si la clé publique de *serveur_nfs* n'est pas présente chez l'utilisateur `snfs`, la commande échouera, il faut donc avoir effectué une connexion au préalable pour éviter ce problème.

Sur le poste client vous pourrez remarquer que :

- le programme `rpc_psrv` est en exécution et appartient à l'utilisateur `snfs`,
- un client SSH appartenant à `snfs` lance l'exécution du programme `rpc_pcl` sur le serveur NFS
- la commande `rpcinfo -p` indique que deux programmes sont enregistrés au niveau du portmapper local (à l'aide de la commande `netstat -ap` on vérifiera que le programme en écoute sur le port `1027` est `rpc_psrv`) :

```
program vers proto   port
 200000    3   udp   1027
 250000    3   udp   1027
```

- sur le serveur le programme `rpc_pcl` est en exécution et appartient à `snfs`.

Pour démonter la partition, il suffit d'exécuter : `/etc/rc.d/init.d/snfs stop`.

Bien que cette configuration permette lors du démarrage du système de monter automatiquement une partition utilisateur (sans aucune intervention de l'administrateur), elle possède un inconvénient majeur. La clé privée de l'utilisateur `snfs` n'étant pas protégée, un intrus réussissant à la subtiliser obtiendrait un accès au serveur NFS. Nous proposons donc la solution suivante qui ne devrait pas être trop pénalisante quant à l'automatisation de toute la procédure :

. Engendrer sur le poste client un couple de clés RSA protégées par une phrase clé. Placer le contenu de `id_rsa.pub` dans le fichier `authorized_keys` de l'utilisateur `snfs` sur *serveur_nfs*.
. Modifier le fichier de configuration `/etc/snfs/`*serveur_nfs* de façon à ce que :
 - le programme `rpc_psrv` soit lancé en tant que `root` (ce qui signifie que le client SSH sera lancé en tant que `root`). Il suffit pour cela de changer la directive `Id snfs` par `Id root`,
 - le client SSH chargé de l'exécution de `rpc_pcl` s'authentifie en tant qu'utilisateur `snfs` sur le serveur NFS. On commentera pour cela les lignes `Host` et `RemoteCommand` et on ajoutera la ligne :

```
SshCommand ssh ssh -c blowfish -x -oFallBackToRsh\ no snfs@serveur_nfs
                              rpc_pcl -P -1 -M 200000:100005 -M 250000:100003
```

. Lancer l'agent SSH pour le `root` et rajouter dans le script `snfs` la ligne `ssh-add /etc/snfs/.ssh/id_rsa`, avant l'appel à `snfsmount` et la ligne `ssh-add -d /etc/snfs/.ssh/id_rsa` avant l'appel à `snfsumount`.

Dorénavant l'exécution de `/etc/rc.d/init.d/snfs start` demandera de fournir la phrase clé associée à la clé `id_rsa` de `snfs` et cette clé sera automatiquement utilisée lors de la connexion au serveur NFS.

Remarques :

. l'option `-M` *num1* `:` *num2* du client proxy RPC signifie que les requêtes RPC émanant du programme numéro *num1* doivent être adressées au programme numéro *num2* enregistré auprès du portmapper local,
. dans le cas d'un démarrage automatique, on peut lancer l'exécution de `ssh-agent` dans le script `snfs`, il faudra cependant récupérer dans un fichier les variables d'environnement créées par ce dernier qui sont indispensables à `ssh-add -d` lors de l'arrêt du script. On pourra rajouter la commande `ssh-agent -k` pour supprimer l'agent lors du démontage de la partition,
. on peut conserver la directive `Id snfs` si dans le script `snfs` les appels à `ssh-add` et `ssh-agent` sont précédés de la commande `su snfs -c` et si on utilise une petite astuce pour la transmission des variables d'environnement liées à `ssh-agent`,
. on peut restreindre les requêtes relayées dans le tunnel en fonction du couple (uid,gid). Si chaque utilisateur crée son propre tunnel, et si ce dernier refuse toute requête non associée à l'uid de l'utilisateur, on obtient alors une authentification forte de chaque utilisateur par le serveur NFS (une attaque reste possible en se plaçant sur le serveur entre le client proxy RPC et le serveur NFS). Cependant ceci ne peut s'appliquer dans le cas où la partition montée est la partition utilisateur,
. SEC RPC peut être utilisé conjointement avec l'automounter AMD, on se référera au fichier `README.NFS` pour plus d'informations.

5.2 Accès indirect

Il suffit d'utiliser un tunnel SSH afin que la demande d'exécution distante du client proxy RPC soit transmise jusqu'au serveur SSH qui se chargera alors de l'émettre en direction du serveur NFS. Afin d'essayer d'automatiser au mieux le procédé sans négliger la sécurité, nous proposons la démarche suivante :

. Sur le poste client, l'administrateur système crée un couple de clés privée/publique non protégées qui servira à la mise en place du tunnel. Pour la suite ces clés seront nommées `id_tunnel` et `id_tunnel.pub`.
. Le contenu de `id_tunnel.pub` est placé sur le serveur SSH dans le fichier `authorized_keys` situé dans le compte utilisateur permettant à l'administrateur système d'accéder au serveur. On rajoute au début de la ligne les directives :
`from="`*poste_client*`",permitopen="`*serveur_nfs*`:22",command="/bin/sleep 100"`
. comme explicité précédemment, on créé un autre couple de clés privée/publique protégées (`id_rsa` et `id_rsa.pub`) qui serviront à l'exécution distante du client proxy RPC. Le contenu de `id_rsa.pub` est rajouté dans le fichier `authorized_keys` de l'utilisateur `snfs` sur *serveur_nfs*.
. L'administrateur système lance sur le poste client son agent SSH et créé le tunnel de communication :

```
client% eval ‘ssh-agent‘
client% ssh -f -i /root/.ssh/id_tunnel -L 2222:poste_client:22 [login@]serveur_ssh sleep 100
```

. Pour que la demande d'exécution distante lancée par `rpc_psrv` soit redirigée dans le tunnel on modifie la directive `SShCommand` comme suit :

```
SshCommand ssh ssh -p2222 -c blowfish -x -oFallBackToRsh\ no snfs@localhost
                              rpc_pcl -P -1 -M 200000:100005 -M 250000:100003
```

. L'administrateur système peut alors exécuter `/etc/rc.d/init.d/snfs start` (modifié comme ci-dessus). Il devra rentrer la phrase clé associée à l'utilisateur `snfs` du serveur NFS.

Remarques :

. à la différence d'un serveur en mode TCP il est inutile ici de créer deux tunnels, puisque le serveur SSH communique avec le serveur NFS via un client SSH (directive `SshCommand`),
. il n'est pas nécessaire que l'utilisateur `snfs` existe sur le serveur SSH. Il suffit que l'administrateur système de *poste_client* possède un compte utilisateur standard sur le serveur pour mettre en place le tunnel,

. le serveur NFS doit aussi héberger un serveur SSH.

6 SHFS

L'installation de SHFS [24] ne concerne que le poste client. La dernière version -en date du 5 septembre 2003- est la 0.32-2. Ce logiciel se compose essentiellement d'un module pour le noyau permettant d'utiliser un nouveau type de système de fichiers pour la commande `mount` : le type `shfs`.

6.1 Accès direct

L'utilisation de la commande `mount -t shfs` revient à exécuter le programme `shfsmount` fourni dans le package, dont la syntaxe générale est : `shfsmount [opts] [`*login*`@]`*serveur_nfs*`[:`*partition*`]` *point_de_montage* `[-o` *mount_opts*`]`. `login` étant le compte utilisateur sur le serveur NFS de l'administrateur système de *poste_client*. L'originalité de SHFS réside dans le fait que toutes les requêtes d'accès distantes sont en fait gérées via un shell SSH. Ainsi il est inutile d'avoir un démon NFS sur le *serveur_nfs*, il suffit qu'un serveur SSH soit actif. Pour la suite nous continuerons cependant -comme dans les paragraphes précédents- à appeler la machine distante *serveur_nfs*. Pour monter la partition `/home` de *serveur_nfs*, l'administrateur système exécute :
`client%shfsmount` *serveur_nfs*`:/home /mnt/home -o rmode=755`. L'option `rmode` précise quels sont les privilèges à donner au point de montage. Par défaut, les privilèges sont définis avec un *ugo* de 700. Lorsque la commande `shfsmount` est exécutée, une fois la phase d'identification effectuée, un accès shell SSH vers le serveur NFS est lancé en tache de fond par le système. A chaque fois qu'un accès au répertoire monté est effectué, le module `shfs.o` l'intercepte et transforme tout appel système en une requête shell. L'ensemble des données transite via deux tubes (pipes), l'un pour la lecture des données et l'autre pour l'écriture. Un système de cache permet de limiter les transferts nécessaires. Pour chaque utilisateur, l'accès aux données de la partition `/mnt/home` se fait traditionnellement par vérification du couple (uid,gid).
Remarques :

. l'administrateur système de *poste_client* doit aussi être l'administrateur système de *serveur_nfs*,
. si la commande exécutée est : `shfsmount` *login@serveur_nfs*`:/home /mnt/home -o rmode=755`, alors toute opération sera associée à l'uid et au gid de *login*.

En utilisant les mécanismes décrits dans les paragraphes précédents, il est très simple de pouvoir automatiser le montage de la partition utilisateur -à partir d'un serveur NFS - au démarrage d'une machine.

Contrôle au niveau utilisateur Dans le cas où les comptes présents sur le serveur NFS ne contiennent que des données annexes pour les utilisateurs de *poste_client*, chacun d'entre eux peut monter dans son propre répertoire de travail les données le concernant (s'il dispose évidemment d'un accès SSH sur la machine). Pour cela, l'administrateur système de *poste_client* doit activer le bit suid pour les commandes `shfsmount` et `shfsumount`. Chaque utilisateur peut alors exécuter (après avoir créé un répertoire `mnt` dans son répertoire de travail) : `shfsmount[` *login*`@]`*serveur_nfs*`:/home/`*login* `mnt`. Le répertoire `mnt` se verra automatiquement attribué un *ugo* de 700. Une question légitime se pose. Que se passe-t-il si l'utilisateur exécute la commande : `shfsmount [`*login*`@]`*serveur_nfs*`:/ mnt` ? L'ensemble du système de fichiers de *serveur_nfs* sera monté dans le répertoire `mnt` de l'utilisateur. Les opérations pouvant être effectuées par ce dernier seront dépendantes de ses droits. Alors qu'un serveur NFS exporte uniquement ce qui doit être visible de l'extérieur, ici c'est l'ensemble du système qui est visible. Ceci n'est pas choquant, étant donné que l'utilisateur a un accès SSH sur le serveur et peut donc prendre connaissance de l'arborescence du serveur, uniquement en lançant un shell.
Cette solution permet donc une authentification forte -par le serveur- de l'utilisateur accédant à une ressource, modulo les réserves que l'on peut émettre quant à la présence d'exécutables suid sur le système.

6.2 Accès indirect

Il suffit de mettre en place un tunnel vers le serveur SSH intermédiaire. Différents scénarios sont envisageables pour la création de ce dernier selon que seul l'administrateur système ou que chaque utilisateur de *poste_client* possède un accès vers le serveur SSH et selon le contexte d'utilisation du serveur NFS (montage des comptes utilisateurs ou accès à des données annexes). A titre d'exemple, dans le cas où la partition `home` est montée au démarrage par l'administrateur de *poste_client* et qu'il est le seul à disposer d'un accès au serveur SSH, le tunnel sera créé comme suit : `ssh -f -L 2222:`*serveur_nfs*`:22 [`*login*`@]`*serveur_ssh* sleep 100, et la partition sera montée en exécutant : `shfsmount -P 2222 localhost:/home /home -o rmode=755`

7 Bilan sur l'utilisation de SSH pour sécuriser NFS

La confidentialité des données transmises sur le réseau est assurée ainsi que leur intégrité.
Le serveur NFS est authentifié par le poste client (sauf dans le cas de l'utilisation d'un seul tunnel lors d'un accès indirect à un serveur NFS en mode TCP).
Il n'y a pas d'authentification du poste client par le serveur NFS. En effet lors de la mise en place du tunnel, seule l'identité de l'utilisateur est contrôlée. Le tunnel (et donc l'accès au serveur NFS) peut être créé à partir de n'importe quelle machine. On peut restreindre l'accès en associant à la clé utilisée pour la création du tunnel, la directive `from="`*poste_client*`"`, ce qui en soit n'est pas une authentification forte du client puisqu'il suffit d'usurper l'adresse IP de ce dernier.
Il n'y a pas d'authentification des utilisateurs -autre que la vérification classique du couple *(uid,gid)*- accédant aux ressources du serveur NFS, seul le créateur du tunnel sécurisé a été authentifié par le serveur (à l'exception de SHFS et SEC RPC dans le cas où ce n'est pas la partition utilisateur qui doit être montée).

8 SFS

SFS (Self-certifying File System) [25] est issu d'une thèse effectuée au MIT par D. Mazières [26], soutenue en Mai 2000. Ce système permet :
. le chiffrement des données entre le poste client et le serveur ainsi que le contrôle de leur intégrité,
. l'authentification du serveur par le poste client,
. l'authentification par le serveur de chaque requête émise par un utilisateur.
. selon l'utilisation, l'authentification du poste client par le serveur.
Pour s'authentifier, chaque serveur possède un couple de clés privée/publique (L_S, K_S). Ceci pose le problème classique de la gestion des clés : comment le poste client récupère-t-il la clé publique du serveur en étant certain de son authenticité (ce problème est d'ailleurs à l'origine d'une attaque sur le protocole SSH). SFS contourne de façon assez élégante ce problème en introduisant la notion de chemin auto-certifié (*self-certifying pathname*). Sur le poste client, tout accès au serveur se fera via un chemin de la forme `/sfs/@`*serveur_nfs*`,`*HostID*`/`*chemin*, où *HostID* est l'image via la fonction de hachage SHA-1 de la chaîne `(1,`*serveur_nfs*,K_S,1,*serveur_nfs*,K_S`)` (le fait de dupliquer le motif d'entrée est une sécurité supplémentaire pour contrer la recherche de collisions). Lorsque le client accède à un chemin de ce type, une requête est émise vers *serveur_nfs*, celui-ci renvoie sa clé K_S et le client peut calculer SHA`-1(1,`*serveur_nfs*,K_S,1,*serveur_nfs*,K_S`)` et comparer le résultat avec *HostID*. Nous verrons un peu plus loin que selon le contexte d'utilisation ce n'est pas réellement ce mécanisme qui est utilisé.
Une question naturelle se pose : par quel moyen peut-on obtenir un *HostID* sans connaître la clé publique d'un serveur ? Dans sa thèse D. Mazières propose plusieurs solutions que nous ne pouvons toutes détailler (les nombreuses fonctionnalités offertes par le protocole SFS justifiant à elles seules un article complet). Nous nous contenterons de présenter par la suite la solution décrite dans la manuel d'utilisation de SFS pour laquelle la procédure d'authentification du serveur diffère de ce qui a été énoncé ci-dessus. Remarquons auparavant que -du fait des propriétés de SHA-1- la quantité *HostID* est unique pour chaque machine. Ainsi un intrus voulant usurper l'identité d'un serveur ne peut envoyer une fausse clé publique (à la différence de SSH lors de la première connexion). L'avantage d'un tel système est que n'importe qui peut mettre en place un serveur SFS sans avoir besoin de faire certifier sa clé publique par une autorité externe quelconque.
Il y a essentiellement 5 programmes qui interviennent dans le protocole SFS : le serveur `sfssd`, le client `sfscd`, l'agent utilisateur `sfsagent` (présent sur le poste client et en exécution pour chaque utilisateur), le démon d'authentification `sfsauthd` (présent sur le serveur). De façon générale, le démon `sfscd` se charge lors de l'accès à un serveur d'authentifier ce dernier et de mettre au point deux clés de session entre le poste client et le serveur, l'une servant à chiffrer les communications du client vers le serveur et l'autre du serveur vers le client. En ce qui concerne le serveur, la construction de ces deux clés nécessite l'utilisation de la clé secrète L_S, ainsi le poste client est assuré que seul le serveur peut déchiffrer la communication.
L'authentification des utilisateurs est effectuée avec un algorithme à clé publique, chaque utilisateur possède donc un couple de clés et le serveur possède un fichier associant à chaque clé publique un couple (uid,gid) définissant les droits associés à cette clé. Lorsqu'un utilisateur accède pour la première fois aux ressources exportées par le serveur, `sfcd` envoie tout d'abord une requête à `sfsagent` qui la signe en utilisant la clé privée de l'utilisateur (ce qui suppose qu'il y ait accès). Le message signé est renvoyé par `sfsagent` à `sfscd` qui le communique au serveur `sfssd`. Ce dernier demande au démon `sfsauthd` de vérifier la signature de la requête. En cas de succès `sfsauthd` renvoie au démon `sfssd` le couple (uid,gid) qui engendre un numéro d'authentification. Afin que le serveur puisse authentifier les requêtes émanants de l'utilisateur, ces dernières seront "marquées" avec ce numéro par `sfscd`. Pour la suite `sfssd` agit comme un client NFS en relayant les demandes au serveur NFS local.
Le protocole SFS a été développé avec pour objectif principal de pouvoir fournir à un utilisateur un accès sécurisé à son serveur de données et ceci depuis n'importe quelle machine (disposant de `sfscd` et `sfsagent`). Si chaque utilisateur stocke dans son répertoire sa clé privée, il doit alors le faire sur tous les postes clients utilisés. Dupliquer des informations secrètes n'est

jamais une bonne chose et rend de plus compliquée leur mise à jour en cas de changement de la clé publique. Bien que ceci soit possible (nous l'avons déjà dit, il existe plusieurs scénarios d'utilisation du protocole), SFS propose une option permettant à l'utilisateur de stocker sa clé privée sur le serveur. Au cas où le serveur serait corrompu, cette clé est stockée de façon chiffrée par une phrase clé et un germe aléatoire de 128 bits en utilisant l'algorithme EKSBLOWFISH [27]. SFS offre un mécanisme permettant à partir de la phrase clé de l'utilisateur :

. d'authentifier le serveur,
. d'authentifier le poste client,
. de récupérer la clé privée chiffrée de l'utilisateur.
. de récupérer la clé publique du serveur.

Étant donné que beaucoup d'utilisateurs choisissent des phrases clé peu fiables et sensibles aux attaques par “dictionnaire”, on peut difficilement faire confiance à un protocole se basant uniquement sur de telles données pour réaliser une authentification. SFS utilise le protocole SRP [28] pour réaliser les quatre opérations précédentes. À partir de données faibles, le protocole SRP construit des quantités secrètes fiables qui seront utilisées comme données d'initialisation pour un algorithme d'authentification basé sur le problème du logarithme discret. Le protocole assure qu'un intrus, observant les quantités échangées sur la ligne, n'a aucun moyen de retrouver les données faibles. Dans le cas de SFS, la phrase clé est transformée via l'algorithme EKSBLOWFISH -et un germe aléatoire- en une instance difficile du logarithme discret. Le protocole SRP a été conçu de façon à effectuer une authentification mutuelle des deux parties l'utilisant.

Afin d'illustrer les propos précédents, nous traiterons un exemple après le détail de la configuration du poste client et du serveur. La version actuelle de SFS est la 0.7.2-1 et peut être obtenue sous la forme de deux packages rpm : `sfs-0.7.2-1` et `sfs-server-0.7.2-1`. Le premier doit être installé sur le client et le serveur.

8.1 Accès direct

Configuration du serveur L'installation des deux packages crée un utilisateur `sfs` sur le système ainsi qu'un répertoire `/etc/sfs`. Bien que `sfscd` corresponde à la partie cliente du protocole, il est préférable de l'exécuter aussi sur le serveur car il gère un mécanisme permettant d'augmenter l'entropie du flux aléatoire utilisé par les différents algorithmes.

Le couple de clé privée/publique du serveur est généré en exécutant : `sfskey gen -P -b 1024 /etc/sfs/sfs_host_key`. L'option `-P` signifie que la clé privée n'est pas protégée par une phrase clé. Il faut ensuite créer le fichier `/etc/sfs/sfsrwsd_config` qui contient des informations concernant les répertoires à exporter. Dans le protocole NFS le client voit une partie de l'arborescence du serveur. Avec SFS il est possible d'exporter une arborescence logique différente de l'arborescence physique en attribuant un nom à chaque répertoire exporté. L'association nom physique/nom logique est contenue dans le fichier `sfsrwsd_config` en utilisant la directive `Export` dont la syntaxe est : `Export` *nom_physique nom_logique*. Le nom logique de la première directive doit être `/` . Les autres doivent correspondre à des répertoires de la racine virtuelle. Le répertoire racine de l'arborescence logique ainsi que tous les répertoires logiques doivent être créés dans le répertoire `/var/sfs` .

Exemple : Supposons que l'on désire exporter les répertoires des utilisateurs `user1`, `user2` et le répertoire `/var/www/html` vers l'arborescence logique `/sfsuser1`, `/sfsuser2` et `/sfsweb`. On commence par créer l'arborescence logique dans `/var/sfs` :

```
serveur_nfs% mkdir /var/sfs/sfsroot
serveur_nfs% mkdir /var/sfs/sfsuser1
serveur_nfs% mkdir /var/sfs/sfsuser2
serveur_nfs% mkdir /var/sfs/sfsweb
```

Le fichier `sfsrwsd_config` correspondant contiendra :

```
Export /var/sfs/sfsroot  /
Export /home/user1       /sfsuser1
Export /home/user2       /sfsuser2
Export /var/www/html     /sfsweb
```

Le démon `sfssd` agissant comme un client NFS standard pour le serveur NFS local, il traduit chaque requête à destination d'une arborescence logique en une requête à destination d'une arborescence physique. Pour cela il utilise l'interface `loopback`, il faut donc que le serveur NFS exporte vers `localhost`, les partitions `/var/sfs/root`, `/home/user1`, `/home/user2` et `/var/www/html`. De plus, `sfssd` s'exécutant sous l'uid `root`, il faut rajouter l'option `no_root_squash` dans le fichier `/etc/exports`. Il ne reste plus qu'à lancer le serveur NFS et le serveur SFS : `/etc/rc.d/init.d/nfs start` ; `/etc/rc.d/init.d/sfssd start`.

Configuration du client : il suffit de lancer `/etc/rc.d/init.d/sfscd start`. Le répertoire `/sfs` est automatiquement créé et le client SFS est en écoute.

Exemple d'utilisation : Pour qu'un utilisateur `user1` puisse accéder à ses ressources sur le serveur SFS, il doit tout d'abord enregistrer sur ce dernier un couple de clé privée/publique qui permettront de l'authentifier :

`user1@`*serveur_nfs*`% sfskey register`

Une phrase clé protégeant la clé privée sera demandée ainsi que 64 caractères aléatoires utilisés pour initialiser un générateur aléatoire. Finalement le mot de passe de `user1` est demandé (au cas où `user1` aurait laissé son terminal accessible et que quelqu'un en profiterait pour changer son couple de clés). `sfskey` calcule à partir de la phrase clé et d'un germe aléatoire, les données nécessaires au protocole SRP qu'il stocke dans le fichier `/etc/sfs/sfs_users`, ainsi que l'uid et le gid de `user1`, sa clé privée chiffrée avec l'algorithme EKSBLOWFISH (en utilisant pour clé de chiffrement le germe aléatoire et la phrase clé) et le germe. Pour accéder à ses données, `user1` exécute à partir du poste client : `sfskey login user1@`*serveur_nfs*, et rentre sa phrase clé (cf. fig. 5, la partie grisée correspond à une communication chiffrée).

Le programme `sfskey` lance l'agent utilisateur `sfsagent`, rentre en contact avec le client `sfcd` et lui demande d'établir une communication sécurisée avec le serveur. Comme explicité précédemment, le serveur envoie sa clé publique K_S et deux clés communes de chiffrement sont mises en place pour la communication client → serveur et serveur → client. Contrairement à ce qui a été présenté en introduction, dans ce contexte le client ne peut authentifier le serveur à partir de K_S puisqu'il ne connaît pas *HostID*. Cependant seul le possesseur de l'inverse de la clé reçue (normalement L_S) est capable à présent de déchiffrer les transactions. On peut donc se trouver dans l'un des deux cas suivants :

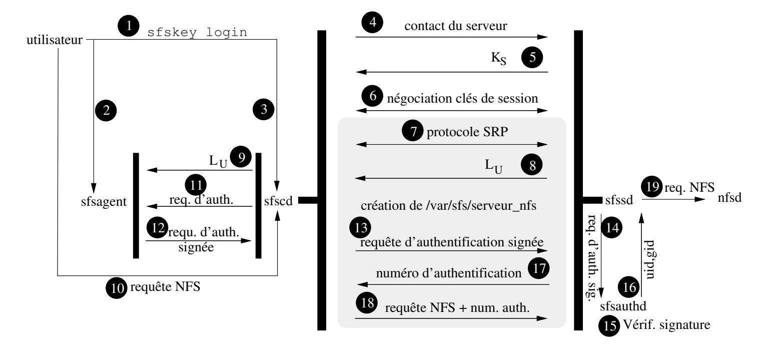


Figure 5 – *Une session sfs*

- un intrus s'est placé en écoute sur la ligne, il ne peut obtenir aucune information,
- un intrus a intercepté la communication et a envoyé sa clé K'_S à la place de K_S, il peut déchiffrer les communications.

Une fois le canal sécurisé en place, les programmes `sfskey` et `sfsauthd` démarrent le protocole SRP. D'après ce qui a été dit précédemment, si le protocole aboutit, le poste client et le serveur se sont authentifiés mutuellement. Dans le cas contraire le client était en contact avec un serveur lui ayant envoyé un fausse clé publique. Le démon `sfsauthd` envoie alors à `sfskey` la clé privée chiffrée de l'utilisateur `user1` et le germe associé. `sfskey` déchiffre la clé (à l'aide du germe et de la phrase clé) et la transmet à `sfsagent`. Ce dernier pourra l'utiliser lorsqu'il s'agira d'authentifier les requêtes de l'utilisateur comme cela a été expliqué dans l'introduction. `sfskey` calcule à partir de K_S la valeur *HostID* (qui est forcément valide) et créé automatiquement le lien : `/var/sfs/`*serveur_nfs* → `/var/sfs/@`*serveur_nfs*`,`*HostID*. Ce lien n'est visible que par l'utilisateur `user1` qui peut accéder à ses données distantes via le chemin `/var/`*serveur_nfs*`/sfsuser1/`.

Remarque : un autre contexte d'utilisation correspond au cas où :

- le lien `/var/sfs/`*serveur_nfs* existe déjà (créé par exemple par l'administrateur système qui certifie que la valeur *HostID* est valide).
- l'utilisateur se connecte toujours à partir du même poste et conserve sa clé privée dans son répertoire de travail sur le poste client, la clé publique correspondant étant sur le serveur.

Dans ce cas, le protocole SRP n'est pas utilisé. L'utilisateur exécute `sfsagent` et `sfskey add` pour transmettre à son agent sa clé privée. Dès qu'il accède à `/var/sfs/`*serveur_nfs/sfsuser1*, le client `sfscd` établit un canal sécurisé et authentifie le serveur (comme explicité dans l'introduction étant donné que *HostID* est connu). L'agent prend ensuite le relais pour authentifier les requêtes de l'utilisateur auprès du serveur. Dans ce contexte le poste client n'est pas authentifié.

8.2 Accès indirect

Le protocole SFS communique en mode TCP. Par défaut le client `sfcd` émet toutes ses requêtes vers le port 4, port d'écoute du serveur. Il n'est pas possible d'indiquer à `sfsagent` et `sfskey` de s'adresser vers un autre port. Ce dernier étant privilégié seul l'administrateur système de *poste_client* peut créer un tunnel SSH entre le poste client et le serveur SSH. Les utilisateurs pouvant se connecter de façon irrégulière au serveur NFS, il est donc conseillé de créer un tunnel permanent : `ssh -f -N -L 4:`*serveur_nfs*`:4 [`*login*`@]`*serveur_ssh*. Pour contacter le serveur NFS, l'utilisateur exécutera : `sfskey login user1@localhost`. Cependant un problème se pose, une fois les phases d'authentification effectuées, `sfskey` va créer un lien du type : `/var/sfs/`*serveur_nfs*. En effet, les informations reçues par `sfskey` contiennent entre autres le nom du serveur. Tout accès via le lien créé dans `/var/sfs` entraînera une requête de connexion directe vers *serveur_nfs* et non pas vers le tunnel. Pour régler ce problème, il faudrait que lors de la communication initiale, `sfskey` reçoive comme nom de serveur l'adresse IP `127.0.0.1` de façon à ce que l'accès à `/var/sfs/127.0.0.1` soit redirigée dans le tunnel. Pour

cela, sur le serveur il faut :
- affecter la variable d'environnement `SFS_HOSTNAME` à `127.0.0.1`,
- générer un nouveau couple de clés privée/publique pour le serveur. La variable `SFS_HOSTNAME` étant définie, le programme n'interrogera pas le DNS pour obtenir le nom du serveur afin de l'associer aux clés générées,
- lancer le serveur `sfssd`.

De plus un client SFS refusant par défaut de requêtes SFS locales, il faut relancer -sur le client- le programme `sfcd` avec l'option `-l` pour que les accès à `127.0.0.1` soient acceptés.

Remarques :
- SFS ne permet pas pour l'instant de monter automatiquement la partition utilisateur au démarrage du système,
- il est possible d'utiliser le mécanisme des ACL (Access Control Lists) sous SFS suite à un travail développé par G. Savvides [29].
- les données échangées entre le serveur et le client sont chiffrées avec l'algorithme ARCFOUR,
- les requêtes NFS sont signées en utilisant le système à clé publique de Rabin-Williams [30] qui est aussi sûr que le RSA si l'on considère que la fonction SHA-1 se comporte comme un oracle aléatoire. De plus, ce système a la particularité suivante : le chiffrement et la vérification d'une signature n'utilisent pas une opération d'exponentiation modulaire (contrairement à RSA).

9 Analyse des performances

Les différents mécanismes de sécurité (notamment les opérations de chiffrement et de déchiffrement) ont nécessairement un impact sur le débit de transmission entre le poste client et le serveur. Pour étudier les dégradations induites nous nous sommes placés dans le contexte de l'accès direct (serveur et client sont reliés directement via le réseau). Le poste client est un PC Pentium III 1Ghz et le serveur NFS un PC Pentium II 350 Mhz. Les deux machines sont connectées à un commutateur 100Mbits/sec. Le test consiste en la copie d'un fichier de 100Mo du poste client vers le serveur.

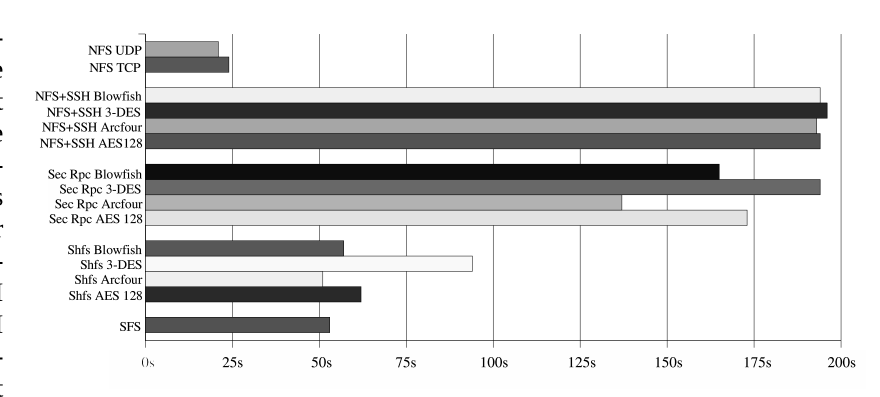


Figure 6 – *Analyse des performances*

10 Autres systèmes

Dans ce paragraphe nous détaillons rapidement deux autres produits -TCFS et CRYPTOFS- qui adoptent une autre stratégie par rapport aux solutions précédentes : les données présentes sur le serveur NFS sont chiffrées. Pour que l'accès à ces dernières soit transparent pour les applications, un module sur le poste client est chargé de chiffrer et déchiffrer les données à la volée. Cependant :
- le projet TCFS n'a plus évolué depuis Avril 2001 et ne fonctionne que sur des postes clients disposant d'un noyau 2.2.17 (Redaht 6.2),
- après contact avec l'un des responsables du projet CRYPTOFS, il s'avère que pour l'instant les sources et les binaires ne sont pas disponibles et qu'un autre projet est en cours de développement.

10.1 Tcfs

TCFS (Transparent Cryptographic File System) a été développé au département d'informatique et applications de l'université de Salerne en Italie. La dernière version est la 3.0b2 [31]. Son installation nécessite de recompiler le noyau du poste client afin d'y inclure le support d'un nouveau système de fichiers : le système `tcfs`. L'administrateur système du poste client monte la partition NFS en spécifiant l'option `-t tcfs` pour la commande `mount`. Le serveur doit être configuré comme un serveur NFS standard. Sur le poste client le fichier `/etc/tcfspwdb` contient pour chaque utilisateur une clé privée chiffrée avec le mot de passe de ce dernier et une variante de l'algorithme MD5. Chaque nouvelle entrée dans ce fichier est créée par l'administrateur avec la commande `tcfsadduser -u` *login*. Chaque utilisateur y inscrit sa clé en spécifiant avec quel algorithme de chiffrement à clé privée elle doit être utilisée : `tcfsgenkey -c 3des`. Dans la dernière version, les algorithmes supportés sont BLOWFISH, 3-DES, RC5 [32] et un algorithme développé par les membres du projet TCFS.
Chaque requête NFS est interceptée par le module `tcfs`. Si une action de chiffrement ou déchiffrement est nécessaire, elle

est effectuée en utilisant la clé de l'utilisateur qui doit être chargée en mémoire. Pour cela l'utilisateur aura exécuté lors de sa connexion la commande `tcfsputkey` qui récupère la clé privée de ce dernier dans le fichier `tcfspwdb` et la déchiffre en utilisant son mot de passe. L'utilitaire `tcfsrmkey` permet de supprimer la clé de la mémoire. L'utilisateur définit quels doivent être les fichiers protégés sur le serveur en exécutant la commande `tcfsflags` qui place l'attribut `X` sur le fichier concerné. Tout fichier ayant un attribut `X` ne sera accessible que si la clé de l'utilisateur est présente en mémoire. Du côté du serveur non seulement le contenu du fichier est illisible mais de plus son nom est chiffré. Un contrôle d'intégrité est intégré dans le fichier, si bien que tout changement du fichier effectué directement sur le serveur (ou lors de son transfert sur le réseau) est détecté par le module `tcfs` au moment des opérations de déchiffrement.

Quelques remarques :

. les données circulant entre le client et le serveur sont chiffrées et leur intégrité est assurée,
. il n'y a pas d'authentification du serveur, ni du client. Cependant seul la personne possédant la clé privée utilisée pour chiffrer les fichiers peut y accéder,
. la sécurité des clés privées stockées sur le poste client ne dépend que du mot de passe des utilisateurs. TCFS permet de charger une clé en mémoire en donnant directement sa valeur sur la ligne de commande, ce qui permet d'utiliser éventuellement une autre méthode de stockage,
. TCFS propose une API de développement permettant de programmer très facilement d'autres algorithmes de chiffrement,
. TCFS possède un mécanisme de partage dans lequel l'accès à un fichier appartenant à n utilisateurs n'est possible que si au moins p d'entre eux ont donné leur autorisation. Pour cela, le fichier est protégé par une clé de groupe et chaque membre possède un morceau de la clé de telle façon que si au moins p d'entre eux se réunissent, la clé entière puisse être reconstruite. Ceci est possible en utilisant un protocole cryptographique classique de partage de secrets basé sur l'interpolation polynomiale [33].

10.2 CryptoFS

Ce protocole a été développé en 2000 par D.P. O'Shanahan à l'université de Dublin [3]. A la différence de TCFS, l'ensemble des données est chiffrée sur le serveur (contenu et nom de fichiers). C'est l'algorithme BLOWFISH avec une clé de 448 bits qui est utilisé. CRYPTOFS ne contrôle pas l'identité de l'utilisateur effectuant une action mais vérifie simplement s'il est autorisé à le faire. Pour cela, un nouveau mécanisme de contrôle d'accès au système de fichiers a été mis en place en utilisant le principe des signatures numériques. Le protocole RSA ainsi que la fonction de hachage MD5 sont exploités de façon classique pour le calcul des signatures (un condensé du message est calculé via la fonction MD5 et ce dernier est traité par l'algorithme RSA). Toutes les opérations de chiffrement et déchiffrement sont effectuées sur le poste client par le module `cryptofs` qui intercepte les requêtes systèmes émises par un processus utilisateur et les transmet ensuite au client NFS. Lorsqu'un nouveau fichier est créé sur la partition importée, le système lui associe automatiquement une clé de chiffrement et un couple de clé privée/publique utilisée pour l'authentification et l'intégrité. Une copie de la clé publique est stockée sur le serveur. Étant donné que les développeurs de CRYPTOFS ne souhaitaient pas modifier la structure du système de fichiers EXT2, plusieurs fichiers intermédiaires sont créés permettant au client NFS de disposer d'informations reliants un uid à un nom de fichier et sa clé publique, ou un uid à un nom de fichier et son couple de clé privée/publique (l'uid correspondant étant alors le propriétaire du fichier). Ainsi dans CRYPTOFS l'utilisateur est libéré de tout problème de gestion de clés.

Lorsqu'un utilisateur émet une requête de lecture pour un fichier :

. le module `cryptofs` engendre une chaîne d'information et la chiffre avec la clé de chiffrement du fichier,
. le résultat est passé au client NFS qui le signe avec la clé publique du fichier (s'il existe une association reliant l'uid au nom de fichier et à sa clé publique),
. la chaîne chiffrée et sa signature sont envoyés au serveur NFS,
. celui-ci calcule à partir de la clé publique du fichier et de la chaîne reçue la signature de cette dernière et la compare avec la signature. Si les deux concordent, la requête est acceptée.

Lorsqu'un utilisateur émet une requête d'écriture pour un fichier :

. le module `cryptofs` engendre une chaîne d'information et la chiffre avec la clé de chiffrement du fichier,
. le résultat est passé au client NFS qui le signe avec la clé privée du fichier (s'il existe une association reliant l'uid au nom de fichier et à sa clé privée),
. la chaîne chiffrée et sa signature sont envoyées au serveur NFS,
. celui-ci calcule à partir de la clé publique du fichier et de la chaîne reçue la signature de cette dernière. Il déchiffre la signature reçue avec la clé publique du fichier et compare le résultat obtenu avec la signature qu'il a calculée. Si les deux concordent, la requête est acceptée.

Une fois la requête accomplie par le serveur, celui-ci signe le résultat avec la clé publique du fichier et renvoie le tout au client NFS qui contrôle la validité de la signature. Le résultat est alors passé au module `cryptofs` qui le déchiffre (si des données sont présentes) avant de le transmettre au processus utilisateur. Ainsi pour effectuer une opération d'écriture sur un fichier il faut connaître sa clé privée, pour une opération de lecture il suffit de posséder sa clé publique. Les données qui transitent sont

chiffrées -soit par le module `cryptofs`, soit parce qu'elles proviennent du serveur- et le fait de signer chaque transaction assure leur intégrité et permet d'authentifier les droits de l'utilisateur.
Le client NFS et le serveur NFS ont été réécrits pour gérer les signatures. L'installation de `cryptofs` nécessite la recompilation du noyau à la fois sur le serveur et le poste client.

11 Nfs 4

Le protocole NFS 4 [2] apporte de nombreuses améliorations dans divers domaines. Nous nous focaliserons uniquement dans ce paragraphe sur les nouveaux mécanismes de sécurité. Le protocole s'appuie toujours sur l'échange de messages RPC pour la communication entre le client et le serveur. NFS 4 dispose des méthodes d'authentification des versions précédentes et en propose une nouvelle intitulée `RPCSEC_GSS` [34]. Cependant cette nouvelle méthode ne se contente pas de réaliser l'authentification des messages, c'est un mécanisme de sécurité à part entière qui repose sur la GSSAPI (Generic Security Service Application Program [35]). Il existe de nombreux procédés permettant de sécuriser les transmissions entre deux intervenants : certains utilisent à la fois la cryptographie à clé publique et la cryptographie à clé privée, d'autres uniquement la cryptographie à clé privée. L'objectif de la GSSAPI est de fournir une interface de programmation générique qui encapsule plusieurs mécanismes de sécurité. Toute session RPC de type `RPCSEC_GSS` est constituée de 3 phases :

- une phase de création de contexte. Le client et le serveur se mettent d'accord sur le mécanisme de sécurité qu'ils vont utiliser, les algorithmes permettant de mettre en œuvre ce mécanisme et le type de service à mettre en place :
  - service d'authentification : les entêtes RPC sont authentifiées avec l'algorithme sélectionné lors de la phase précédente,
  - service d'intégrité : les entêtes RPC sont authentifiées. L'intégrité des données est vérifiée.
  - service de confidentialité : inclus le service d'intégrité et chiffrement des données RPC.
- une phase d'échange de données correspondant à la communication entre le client et le serveur,
- une phase de destruction de contexte.

Remarque : lors de la mise en place de la phase de création de contexte, le serveur renvoie une valeur indiquant le nombre maximum de requêtes pouvant être traitées durant l'existence du contexte.
Lorsqu'un client NFS contacte pour la première fois un serveur, il envoie la requête RPC `SECINFO` afin d'obtenir la méthode d'authentification devant être utilisée pour la suite de la communication. C'est à cette étape que le serveur peut choisir entre `RPCSEC_GSS` et les autres alternatives. Le serveur a la possibilité d'associer à chaque système de fichiers qu'il exporte une méthode d'authentification. Cette première communication se fait via un canal sécurisé pour éviter qu'un attaquant -actif sur la ligne- ne modifie le niveau de sécurité requis.
Actuellement trois mécanismes de sécurité sont disponibles dans le protocole NFS 4 :

- KERBEROS V5 [36] utilisant uniquement la cryptographie à clé privée. L'algorithme imposé pour le chiffrement est le DES, il est donc vivement conseillé d'attendre une mise à jour de la RFC 1964 prenant en compte l'AES. Un draft internet est en cours de rédaction à ce sujet [37].
- LIPKEY [38] : les algorithmes d'authentification, d'intégrité et de chiffrement sont négociés entre le client et le serveur. Cependant certains sont recommandés et, si un serveur ne les propose pas, le client peut refuser la communication (et vice-versa). Pour authentifier le serveur, le client vérifie son certificat en le comparant avec une liste issue de serveurs de certification. Le client génère alors une clé de session et la chiffre -ainsi que l'identifiant et le mot de passe de l'utilisateur- avec la clé publique du serveur. Le tout est envoyé au serveur qui contrôle l'identité de l'utilisateur en vérifiant l'identifiant et le mot de passe associé. La clé de session est utilisée pour chiffrer le reste des transactions. Pour le contrôle d'intégrité, l'algorithme HMAC-MD5 est recommandé. Pour l'authentification l'utilisation du RSA avec MD5 est conseillé. L'algorithme CAST [39] en mode cbc est suggéré pour le chiffrement, ce dernier résistant aux cryptanalyses différentielles et linéaires.
- SPKM-3 qui consiste en un sur-ensemble de LIPKEY. Ce mécanisme est utilisé dans le cas où le client réalise un accès anonyme (il ne dispose donc ni d'un identifiant ni d'un mot de passe) ou bien lorsque le client dispose d'un certificat (l'authentification n'est alors pas réalisée par mot de passe).

NFS 4 utilisant la GSSAPI, il est très facile d'y rajouter d'autres mécanismes de sécurité selon les contextes d'utilisation du serveur. Actuellement, il n'existe pas de versions officielles d'un client et d'un serveur NFS 4 pour Linux. Une version de développement est disponible dans la série 2.5 du noyau et devrait faire partie intégrante de la version 2.6. Cependant, il est préférable d'utiliser les versions de développement mises au point par le CITI (Center for Information Technology Integration) de l'université du Michigan dans lesquelles de nombreux bugs de la version proposée dans le noyau 2.6 ont déjà été corrigés. De plus, les packages fournis peuvent être installés sur un système pourvu d'un noyau 2.4. Cependant tous les mécanismes de sécurité ne sont pas présents.

Références

[1] B. Callaghan et B. Pawlowski et P. Staubach. Nfs version 3 protocol specification. *RFC 1813*, Juin 1995.

[2] S. Shepler et B. Callaghan et D. Robinson et R. Thurlow et C. Beame et M. Eisler et D. Noveck. Network file system (nfs) version 4 protocol. *RFC 3530*, Avr. 2003.

[3] D.P. O'Shanahan. Cryptofs : Fast cryptographic secure nfs. Mémoire de D.E.A., Univ. of Dublin, 2000.

[4] G. Cattaneo et L. Catuogno et A. Del Sorbo et P. Persiano. The design and implementation of a transparent cryptographic file system for unix. Dans *USENIX Annual Technical Conference 2001*, Boston MA, Juin 2001.

[5] R. Srinivasan. Rpc : Remote procedure call specification version 2. Août 1995.

[6] W. Diffie et M.E. Helmann. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6) :644–654, Novembre 1976.

[7] B.A. LaMachhia et A.M. Odlyzko. Computation of discrete logarithms in prime fields. *Design, Codes and Cryptography*, 1 :46–62, 1991.

[8] NBS FIPS PUB 46. Data encryption standard. Rapport technique, National Bureau of Standard, U.S. Department of Commerce, 1977.

[9] E. Biham et A. Shamir. Differential cryptanalysis of des-like cryptosystems. Dans *Advances in Cryptology - CRYPTO'90*, pages 2–21, Berlin, 1991.

[10] M. Matsui. The first experimental cryptanalysis of the data encryption standard. Dans *Advances in Cryptology - CRYPTO'94*, pages 1–11, Berlin, 1994.

[11] Electronic Frontier Foundation. *Cracking DES : Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly and Associates, 1998.

[12] T. Ylönen. Secure login connections over the internet. Dans *6th USENIX Security Symposium*, pages 37–42, San Jose, Juillet 1996.

[13] R.L. Rivest et A. Shamir et L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, Février 1978.

[14] NBS FIPS PUB 186. Digital signature standard. Rapport technique, National Institute of Standards and Technology, U.S. Department of Commerce, Mai 1993.

[15] Openssh. *http ://www.openssh.org/*.

[16] Arjen K. Lenstra et Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology : the journal of the International Association for Cryptologic Research*, 14(4) :255–293, 2001.

[17] ANSI X9.17(Revised). Rapport technique, American National Standard for Financial Institution Key Management (Wholesale), 1985.

[18] J. Daemen et V. Rijmen. The block cipher rijndael. Dans *Smart Card Research and Applications*, numéro LNCS 1820, pages 288–296, Novembre 2000.

[19] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). Dans *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204, Cambridge, 1994.

[20] K. Kaukonen et R. Thayer. A stream cipher encryption algorithm, arcfour. *Internet draft (draft-kaukonen-cipher-arcfour-03), Network working group*, July 1999.

[21] Proposed revision of federal information processing standard (fips) 180, secure hash standard (dss). *Federal Register*, 59(131) :35317–35318, 1994.

[22] R.L. Rivest. The md5 message digest algorithm. *RFC1321*, 1992.

[23] J. Bowman. Secure nfs and nis via ssh tunnel. *http ://www.math.ualberta.ca/imaging/snfs/*.

[24] (secure) shell filesystem linux kernel module (shfs/sshfs). *http ://shfs.sourceforge.net/*.

[25] Self-certifying file system. *http ://www.fs.net/sfswww/*.

[26] D. Mazières. *Self-certifying File System*. Thèse de doctorat, Massachusetts Institute of Technology, 2000.

[27] N. Provos et D. Mazières. A future-adaptable password scheme. Dans *Proceedings of the 1999 USENIX, Freenix Track*, Monterey, CA, 1999.

[28] T. Whu. The secure remote password protocol. Dans *Proceedings of the 1998 Internet Society Neywork and Distributed Security Symposium*, pages 97–111, San Diego, CA, 1998.

[29] G. Savvides. Access control lists for the self-certifying filesystem. Mémoire de D.E.A., Massachusetts Institute of Technology, 2002.

[30] H.-C. Williams. A modification of the rsa public-key encryption procedure. *IEEE Transactions on Information Theory*, 26(6) :726–729, Novembre 1980.

[31] Transparent cryptographic file system. *http ://www.tcfs.it/*.

[32] R.L. Rivest. The rc5 encryption algorithm. Dans *K.U. Leuven Workshop on Cryptographic Algorithms*, Belgique, 1995.

[33] A. Shamir. How to share a secret. *Communications of the ACM*, 24(11) :612–613, Novembre 1979.

[34] M. Eisler et A. Chiu et L. Ling. Rpcsec gss protocol specification. Sep. 1997.

[35] J. Linn. Generic security service application program interface, version 2. *RFC 2743*, Javn. 2000.

[36] J. Linn. The kerberos version 5 gss-api mechanism. *RFC 1964*, Juin 1996.

[37] K. Raeburn. Aes encryption for kerberos 5. *draft-raeburn-krb-rijndael-krb-05.txt*, Juin 2003.

[38] M. Eisler. Lipkey - a low infrastructure public key mechanism using skpm. *RFC 2847*, Juin 2000.

[39] C.M. Adams et S.E. Tavares. Designing s-boxes for ciphers resistant to differential cryptanalysis. Dans *Proceedings of the 3rd Symposium on State and Progress Research in Cryptography*, pages 181–190, Rome, Italie, 1993.